UNIVERSITE PAUL SABATIER

TOULOUSE III

Introduction au langage C

L2 IUP AISEM

2004/2005

J.M. ENJALBERT

1 Introduction

Un langage de programmation est un intermédiaire entre l'homme et la machine. Il permet de faire effectuer des tâches à une machine programmable en utilisant des concepts proches de la pensée humaine.

Le langage C fait partie de la famille des langages de programmation fonctionnels de haut niveau (comme Pascal, Ada, Fortran, etc...). Il a été développé dans les années 70 par Kernighan et Ritchie aux laboratoires Bell d'AT &T. Il est l'aboutissement de deux langages: BPCL développé par Richards et B développé par Thompson (d'où le nom de langage C). Leur objectif premier était de réécrire en langage évolué le système d'exploitation UNIX de manière à assurer sa portabilité. Il s'est montré en fait plus polyvalent et est utilisé aussi bien pour écrire des applications de calcul scientifique que de gestion.

C'est un langage très utilisé dans l'industrie car il cumule les avantages d'un langage de haut-niveau (portabilité, modularité, etc...) et ceux des langages assembleurs proches du matériel mais plus difficiles à utiliser pour des projets importants.

2 Eléments de codage des informations

Un ordinateur ne manipule que des données binaires: un transistor est bloqué ou passant. Les données qu'un être humain manipule sont beaucoup plus complexes qu'il s'agisse de nombres, de chaînes de caractères ou d'autres entités. Utiliser une machine pour manipuler ces données implique donc une étape de *codage*.

2.1 Bases de numération

La base de numération que l'on manipule quotidiennement est la base décimale (base 10). Celle-ci est cependant assez mal adaptée a la manipulation des informations dans un ordinateur. Celui-ci ne connaissant que les valeurs binaires (que l'on représente généralement par les symboles 0 et 1), la base naturelle est la base 2.

2.2 Entiers naturels - Numération positionnelle

La numération positionnelle consiste à représenter les nombres à l'aide d'un ensemble de symboles (chiffres en représentation décimale) en leur donnant un poids qui dépend de leur position.

2.2.1 Représentation décimale (base 10)

L'ensemble des symboles est constitué des dix chiffres 0,1,...,9. Le poids d'un chiffre dépend de sa position: 1 (soit 10^0) pour le chiffre le plus à droite, $10^n - 1$ pour le nième chiffre en allant vers la gauche.

2.2.2 Représentation binaire (base 2)

Deux symboles: 0 et 1.

2.2.3 Représentation hexadécimale (base 16)

16 symboles: les chiffres 0 à 9 plus les lettres A à F.

2.2.4 Passage d'une représentation à l'autre

- Passage décimal → binaire: divisions successives par 2
- Passege décimal → hexadécimal: divisions par 16
- Passage binaire → hexadécimal: regroupement des bits par 4

2.2.5 Entiers naturels dans un calculateur

Différents types (en C):

- `unsigned char`: Entiers naturels sur 1 octet (8 bits) (0 à 255)
- `char`: Entiers relatifs sur 1 octet (-128 à +127)
- `unsigned short int`: Entiers naturels sur 2 octets (0 à 65535)
- `short int`: Entiers relatifs sur 2 octets (-32768 à 32767)
- `unsigned long int`: Entiers naturels sur 4 octets (0 à 4 294 967 295)
- `long int`: Entiers relatifs sur 4 octets (-2 147 483 648 à 2 147 483 647)
- `unsigned int` et `int`: la longueur dépend du compilateur (et du processeur): 2 octets ou 4 octets.

2.3 Codage des entiers sur un octet

Code: $A = a_7a_6a_5a_4a_3a_2a_1a_0$ avec $a_i \in \{0, 1\}$

- Entiers naturels

$$N(A) = \sum_{i=0}^{i=7} a_i 2^i$$

- Signe et valeur absolue

$$S(A) = (-1)^{a_7} \sum_{i=0}^{i=6} a_i 2^i$$

- Technique de l'excédent

$$X_e(A) = \sum_{i=0}^{i=7} a_i 2^i - e = N(A) - e$$

- Complément à 2

$$Z(A) = -a_7 2^7 + \sum_{i=0}^{i=6} a_i 2^i$$

Code A	Hexa	$N(A)$	$S(A)$	$X_{128}(A)$	$Z(A)$
0000 0000	0x00	0	0	-128	0
0000 0001	0x01	1	1	-127	0
0000 0010	0x02	2	2	-126	0
...	...	...	...	...	...
0111 1101	0x7D	125	125	-3	125
0111 1110	0x7E	126	126	-2	126
0111 1111	0x7F	127	127	-1	127
1000 0000	0x80	128	0	0	-128
1000 0001	0x81	129	-1	1	-127
1000 0010	0x82	130	-2	2	-126
...	...	...	...	...	...
1111 1101	0xFD	253	-125	125	-3
1111 1110	0xFE	254	-126	126	-2
1111 1111	0xFF	255	-127	127	-1

2.4 Réels en virgule fixe

$$\Phi_m(A) = -a_{n-1}2^{n-1-m} + \sum_{i=0}^{n-2} a_i 2^{i-m} = 2^{-m} Z(A)$$

2.5 Réels en virgule flottante - Norme IEEE P754

Simple précision (32 bits):

31	30 ... 23	22 ... 0
S	E	F

$$R(A) = (-1)^S (1 + \Phi_{23}(F)) 2^{X_{127}(E)}$$

3 Généralités sur le langage C

3.1 Fichier source et exécutable

Un fichier peut être défini comme une entité regroupant un ensemble d'informations, stockée sur un support physique (disque par exemple) et manipulable grace à un système d'exploitation.

On peut distinguer différents type de fichiers:

- Des fichiers exécutables (applications).Un fichier exécutable contient du code directement interprétable et exécutable par le processeur. Sous dos, ces fichiers ont l'extension[1] .exe ou .com. Sous UNIX c'est une propriété donné au fichier qui indique au système qu'il s'agit d'un exécutable.
- Des fichiers binaires: ils contiennent du code machine mais ne sont pas directement exécutables: fichiers objets.
- Des fichiers texte (ASCII). Ces fichiers contiennent uniquement des codes dit ASCII (codés sur 7 bits). C'est le type de fichiers le plus portable.
- Des fichiers liés a une application particulière: fichiers musicaux (mp3, waw,...) d'images (jpeg, gif, tiff, png, etc...) de vidéo, liés à un traitement de texte (word...).

1. on apelle extension d'un fichier les caractères suivant le dernier point de son nom.

L'objectif d'un programmeur est bien sur d'arriver à générer (puis exécuter) un fichier exécutable. Ceci passe par plusieurs étapes. [2] que nous allons décrire dans le cas d'un programme en langage C.

- La première étape consiste à écrire le programme (on parle de source) dans un fichier texte à l'aide d'un éditeur. En C, on donne l'extension .c à ce fichier. Ce programme est bien evidemment incompréhensible par la machine.
- La deuxième étape est l'étape de précompilation. Elle consiste a traiter les directives de compilation (comme l'inclusion de fichiers d'entête de bibliothèques). Elle génère un fichier texte qui est encore un fichier source en C.
- L'étape suivante est la compilation. Elle consiste a transformer les instructions du programme en langage compréhensible par le processeur (langage machine). Elle génère un fichier binaire dit fichier objet.
- La dernière étape consiste à effectuer l'édition de liens. Le code généré à la compilation est complété par le code des fonctions des bibliothèques utilisées. C'est seulement après cette étape que l'on génère un fichier exécutable.

en général, ces étapes sont transparentes pour l'utilisateur. Par exemple sous UNIX on appellera le compilateur à l'aide d'une commande du type: `gcc -o prog prog.c` qui créera l'exécutable `prog` à partir du source `prog.c`.

3.2 Structure d'un programme C

Un programme source en langage C est une collection de constantes, de variables et de fonctions pouvant être définies dans plusieurs fichiers séparés. Une fonction nommée `main` doit exister de manière unique dans cet ensemble. Elle correspond au programme principal exécuté par la machine et faisant appel aux autres fonctions. Chaque fonction est constituée d'un entête et d'un bloc délimité par les caractères { et }. Un bloc est contitué de déclarations de variables et d'une suite d'instructions.

Voici un exemple permettant d'illustrer la structure d'un programme C:

```
/*
  Calcul des puissances successives de 2
 */
#include<stdio.h>
/*-----------------------------------------------------------*/
long int puissance(long int X, int Y); /* calcule x^y */
/*-----------------------------------------------------------*/
int main()
{
  long int resultat;
  int i;

  for (i=0; i<=N; i++)
         {
                resultat=puissance(2,i);
                printf("2 puissance %d =%ld\n",i,resultat);
         }
  return 0;
}
/*-----------------------------------------------------------*/
```

2. Ceci est vrai pour un langage dit compilé

```
long int puissance(long int X, int Y)
{
  int i;
  long int P=1;

  for (i=0; i<Y; i++)
        P=X*P;

  return P;
}
```

On distingue dans ce programme une première zone contenant des directives de compilation (lignes précédées du caractère #).

On trouve ensuite un entête de fonction (déclaration).

Suit le code des fonctions composant le programme en commencant par la fonction `main` correspondant au programme principal.

Chaque fonction possède un entête composé de:

- son type
- son nom (identificateur)
- une liste de paramètres (qui peut être vide)

Cet entête est suivi d'un bloc (délimité par les caractères { et }. Dans ce bloc on trouve une zone de déclaration de variables (type suivi du nom de la variable). Puis la suite d'instructions que doit exécuter la fonction.

On distingue les instructions simples (se terminant par le caractère ;) des instructions composées délimitées par les caractères { et }. Les instructions du préprocesseur (directives de compilation) sont précédées du caractère # et les commentaires sont encadrés par les délimiteurs: /* et */.

4 Eléments du langage

4.1 Identificateurs

Un identicateur est le nom donné aux différentes composantes d'un programme: variables, fonctions,... Il doit être formé d'au maximum 31 caractères pris dans l'ensemble composé des lettres a-z et A-Z[3], des chiffres 0-9 et du caractère spécial _ (soulignement).

4.2 Les données dans un programme

Un programme manipule des données de différents types sous la forme de constantes ou de variables. Les principaux types de base sont:

- `int` nombre entier
- `char` caractère
- `float` nombre réel simple précision
- `double` nombre réel double précision

Ces types peuvent par ailleurs être *qualifiés* par les qualificateurs suivants:

- `short` format plus court que le type de base
- `long` format plus long que le type de base
- `signed` nombre signé

3. En C comme en UNIX, on distingue les majuscules des minuscules.

– `unsigned` nombre non signé

Ces qualificateurs s'appliquent essentiellement au nombres entiers. Par exemple `unsigned short int` est le type d'un entier non signé codé sur 16 bits.

Il existe aussi un type particulier: `void` qui représente en fait une absence de type.

4.2.1 Variables

Les données d'un programme sont stockées en mémoire et sont manipulées par l'intermédiaire d'identificateurs de variables. Par exemple la déclaration: `int N` réserve de l'espace en mémoire pour un entier dont l'identificateur (nom qui permet de le manipuler) est `N`.

4.2.2 Constantes

- Une constante entière peut s'écrire dans les systèmes décimal et hexadécimal. Exemples: `12 0x35`. Par défaut elle est de type `int`.
- Une constante réelle est un nombre exprimé en base 10 contenant un point décimal et éventuellement un exposant séparé du nombre par la lettre `e` ou `E`. Exemples: `10. 5.4 3.5e-3`
- Une constante caractère est assimiliée à un entier codé sur un octet dont la valeur correspond au code ASCII du caractère. Elle est constituée d'un caractère entre apostrophes (quotes). Exemples: `'z' 'A'`
- Une constante chaîne de caractères est une suite de caractères entre guillemets (double-quote). Exemple: `"une chaîne"`. En mémoire cette suite de caractères se termine par le caractère `NULL`. Il s'agit en fait d'un tableau de caractères, notion que l'on abordera plus loin.

4.2.3 Type énuméré

Le mot clé `enum` permet de créer des types *énumérés* c'est à dire des sous ensembles d'entiers.

Par exemple

```
enum booleen {FALSE, TRUE);
```

défini un type `enum booleen` qui associe à l'identificateur `FALSE` la valeur 0 et a l'identificateur `TRUE` la valeur 1. On peut ensuite déclarer des variables de ce nouveau type et leur affecter des valeurs:

```
enum booleen test;
test=FALSE;
```

Autres exemples:

```
enum couleurs {bleu, blanc,rouge};
enum mois {jan=1, fev, mars, avr, mai, juin, juil, aout, sept, oct, nov, dec};
```

4.3 Expressions et Opérateurs

Deux types d'opérateurs peuvent être appliqués aux données (ou opérandes):

- Les opérateurs *unaires* qui s'applique à un seul opérande (exemple: la négation)
- Les opérateurs *binaires* qui possèdent deux opérandes (exemple: l'addition)

Une expression est une combinaison d'opérateurs et d'opérandes qui conduit à un résultat unique. Le type du résultat dépend des types des opérandes la conversion se faisant dans l'ordre suivant: char → int → float → double

Par exemple: si `i` est de type `int` et `x` de type `float`, le résultat de `i*x` sera de type `float`.

4.3.1 Opérateurs arithmétiques

Il s'agit des opérateurs binaires d'addition (+), de soustraction (-) de division (/) et de multiplication (*) et des opérateurs unaires + et -. A cela s'ajoute l'opérateur % qui s'applique à deux entiers et donne le reste de la division entière.

Enfin les opérateurs d'incrémentation (++) et de décrémentation (−−) complètent cet ensemble.

4.3.2 Opérateurs relationnels

Il s'agit des opérateurs d'égalité (==), d'inégalité (!=) ou de comparaison: $<, <=, >, >=$ (respectivement: inférieur, inférieur ou égal, supérieur, supérieur ou égal).

Ces opérateurs renvoient une valeur entière de type `int` égale à 0 (faux) ou a 1 (vrai) suivant le résultat de l'opération (les variables booléennes n'existent pas de manière explicite en C).

4.3.3 Opérateurs logiques

Ces opérateurs s'appliquent à des opérandes qui ont la valeur logique FAUX s'ils valent 0 et la valeur logique VRAI sinon.

Il s'agit de la négation (!) du ET logique (&&) et du OU logique (||).

Exemple: si A vaut 1 et B vaut 0, alors A && B vaut 0.

4.3.4 Opérateurs de manipulation de bits

Le langage C permet de manipuler les bits des opérandes de type entier. Les opérations suivantes se font *bit a bit*.

ET (&), OU inclusif (|), OU exclusif (ˆ)

$<<$ réalise un décalage vers la gauche ($a << n$: décalage de n bits)

$>>$ réalise un décalage vers la droite. Dans le cas d'un entier non-signé, les bits les plus a gauche sont remplis par des zéros. S'il s'agit d'un entier signé, ils sont remplis par la valeur du bit de signe.

˜effectue le complément à 1 de la chaîne de bits.

Exemples: soit `A=26=0x1A=00011010` et `B=43=0x2B=00101011`

- l'expression `A & B` vaut `00001010=0x0A=10`
- l'expression `A | B` vaut `00111011=0x3B=59`
- l'expression `A ^B` vaut `00110001=0x31=49`
- l'expression `A << 2` vaut `01101000=0x68=104`
- l'expression `A >> 2` vaut `00000110 =0x06=6`
- l'expression `~A` vaut `11100101=0xE5=229`

4.3.5 Priorités des opérateurs

L'ordre de priorité des opérateurs vus précédemment est résumé dans le tableau suivant en commencant par les opérateurs les plus prioritaires. Les opérateurs situés sur la même ligne ont le même ordre de priorité. Les opérateurs unaires (comme + et -) ont priorité sur leur forme binaires. L'utilisation de parenthèses permet de lever toute ambiguité dans l'ordre d'évaluation, les expressions entre parenthèses étant les plus prioritaires.

Opér. Unaires	+	-	++	--	~	!
Opér. Binaires	*	/	%			
	+	-				
	>>	<<				
	<	<=	>	>=		
	==	!=				
	&					
	^					
	|					
	&&					
	||					

4.4 Instructions

Les instructions d'un programme effectuent des opérations de manière séquentielle. Quand une instruction a terminé son travail,c'est la suivante qui prend la suite. Les instructions sont séparées par le symbole ;. Une expression terminée par un ; devient une instruction.

Un bloc (délimité par les symboles { et }) est une instruction composée

Les différents types d'instructions élémentaires sont décrits ci-dessous.

4.4.1 Affectation

L'affectation simple (=) affecte à la variable située à gauche du =, la valeur de l'expression située à droite. Ex: `A=2*5+3` affecte la valeur 13 à la variable A. L'affectation est par ailleurs aussi une expression dont la valeur est celle du membre de droite de l'affectation. Ex: l'expression `A=2*5+3` vaut 13 et `B=A=2*5+3` affecte donc la valeur 13 à la variable B (et toujours 13 à A).

L'affectation combinée `+=, -=, *=, /=, etc..` est un raccourci d'écriture qui affecte à la variable de gauche le résultat de l'opération indiquée avant le signe = entre l'expression de droite et la variable elle-mème. Ex: `X+=3` est équivalent à écrire `X=X+3`.

4.4.2 Alternative

Syntaxe: `if (expression) instruction1; [else] instruction2;`

Si `expression` est vraie, alors `instruction1` est exécutée sinon c'est `instruction2` qui est exécutée.[4]

4.4.3 Itérations

- Boucle `while`: syntaxe:

```
while (expression) instruction;
```

  L'expression est évaluée. Si sa valeur est VRAIE , l'instruction est alors exécutée. Puis l'expression est a nouveau évaluée.
- Boucle `do-while`: syntaxe:

```
do instruction while (expression);
```

4. Les crochets ne sont là que pour indiquer que le `else` est optionnel.

L'instruction est exécutée puis l'expression est évaluée. Si sa valeur est VRAIE ,l'instruction est a nouveau exécutée.

– Boucle `for`: syntaxe:

```
for (expres1; expres2; expres3) instruction
```

L'expression 1 est évaluée (initialisation). L'expression2 est elle aussi évalué (test de fin de boucle). Si elle est VRAIE l'instruction est exécutée puis l'expression3 est évaluée (action associée à une itération). L'expression2 est alors à nouveau évaluée, etc...
Formellement ceci est équivalent à:

```
expres1;
while (expres2)
   {
    instruction;
    expres3;
   }
```

En pratique on l'utilise pour créer une boucle dont on connait le nombre de fois ou elle doit être exécutée. Typiquement on aura:

```
for (i=0; i<N; i++)
{
... /* instructions à réaliser N fois */
}
```

4.4.4 Choix multiple

L'instruction *switch* regarde si la valeur d'une expression fait partie d'un certain ensemble de constantes entières et effectue les traitements associés à la valeur correspondante.

Syntaxe:

```
switch(expression)
{
case expression-constante1: instructions1;
case expression-constante2: instructions2;
...
default: instructions;
}
```

Chaque cas possible est étiqueté par une ou plusieurs constantes entières. L'expression est évaluée. Si elle correspond a une expression-constante, l'exécution commence par les instructions associées a cette étiquette.

L'instruction *break* permet de sortir immédiatement du switch. En effet lorsque le traitement d'un cas est terminé, l'exécution continue par le cas suivant. Pour différencier les divers cas il est donc nécessaire de terminer chacun d'entre eux par l'instruction *break*.

4.4.5 Break

L'instruction *break* que nous venons de voir permet de sortir, non seulement d'un switch, mais aussi des boucles do, while et for.

4.5 Entrées/Sorties standards

La manière standard de rentrer des données dans un programme est d'utiliser le clavier et la manière standard de récupérer des résultats consiste a utiliser l'écran.

En fait il s'agit d'un cas particulier d'entrées/sorties sur des fichiers que l'on verra par la suite.

Les fonctions d'entrées/sorties ne font pas partie du langage mais sont des fonctions d'une bibliothèque. Celle-ci est cependant standard et l'utilisation des fonctions de cette bibliothèque laisse le programme parfaitement portable.

Pour accéder aux fonctions de cette bibliothèque il est nécessaire d'inclure le fichier `stdio.h` grace à une directive de compilation placée en début de programme:

```
#include <stdio.h>
```

4.5.1 Entrées/Sorties de caractères

La fonction de base qui permet de lire un caractère au clavier est `int getchar()`. Les entrées au clavier sont en fait rangées dans un "buffer". C'est l'apparition du caractère "Enter" qui permet d'accéder aux caractères tapés, y compris le caractère "Enter". Ceci rend l'utilisation de *getchar()* délicate.

La fonction qui permet d'écrire un caractère à l'écran est `int putchar(int)`. Par exemple, le petit programme suivant permettra d'écrire le message `Bonjour` à l'écran.

```
#include <stdio.h>

int main()
{
  putchar('B');
  putchar('o');
  putchar('n');
  putchar('j');
  putchar('o');
  putchar('u');
  putchar('r');
  putchar('\n');

  return 0;
}
```

4.5.2 Entrées/Sorties formatées

Les deux fonctions précédentes ne sont pas très pratiques et on utilise généralement des fonctions de plus haut niveau pour effectuer les entrées clavier et les sorties écran.

La fonction **printf** permet d'écrire à l'écran des chaînes de caractères et des données numériques. Sa syntaxe est la suivante:

```
int printf(char *format, arg1, arg2, arg3, ...)
```

Elle admet un nombre quelconque d'arguments.

*char *format* est une chaîne de caractères qui contrôle le format d'affichage des arguments de printf. Elle contient des caractères ordinaires affichés tel quel à l'écran et des spécifications de conversion dont chacune porte sur un argument de printf. Ces spécifications commence par le caractère % et se termine par un caractère de conversion.

Les principaux caractères de conversion sont:

- `d` ou `i` pour un entier.

- c pour un caractère.
- s pour une chaîne de caractères.
- f pour un réel en virgule fixe
- e pour un réel en notation scientifique

Entre le % et le caractère de conversion on peut rajouter (dans l'ordre):

- Le signe - pour cadrer l'argument à gauche
- Un nombre qui précise la largeur minimale du champ d'impression.
- un point suivi d'un nombre qui précise le nombre de chiffres après la virgule a afficher (pour un réel).
- la lettre l pour un *long int* ou h pour un *short int*

Exemples:

```
#include<stdio.h>

int main()
{
  int i=123;
  double X=3456.34;

  printf("Bonjour\n");
  printf("Si je multiplie %d par %f j'obtiens un réel égal à %e\n",i,X,i*X);
  printf("Soit %4d * %8.2f = %12.4e \n",i,X,i*X);

  return 0;
}
```

Affichera à l'écran:

```
Bonjour
Si je multiplie 123 par 3456.340000 j'obtiens un réel égal à 4.251298e+05
Soit  123 *  3456.34 = 4.251298e+05
```

La fonction *printf* renvoie par ailleurs le nombre de caractères affichés.

La fonction **scanf** fournit des possibilités de conversions semblables à printf mais pour des entrées.Sa syntaxe est:

```
int scanf(char *format, &arg1, &arg2, &arg3, ...)
```

Les arguments (autres que format) de *scanf* doivent être des *adresses* de variables (ou des pointeurs).

La chaîne de caractères *format* contient les spécifications de conversion qui indiquent comment interpréter les valeurs rentrées. A chaque argument de scanf doit correspondre une spécification de conversion constituée du caractère % suivi d'une lettre.

- d ou i pour un entier sous forme décimale (int)
- u pour un entier non signé
- ld ou li pour un entier long (long int)
- hd ou hi pour un entier court (short int)
- x pour un entier sous forme hexadécimale
- e, f ou g pour un réel simple précision (float)
- lf, le ou lg pour un réel double précision (double)
- c pour un caractère

– s pour une chaîne de caractères [5]

Exemple:

```
#include <stdio.h>

int main()
{
  char chaine[10];
  double reel;
  int entier;

  scanf("%s %lf %i",chaine,&reel,&entier);
  printf("|%s|%f %i\n",chaine,reel,entier);
  return 0;
}
```

La fonction scanf retourne le nombre de données correctement affectées.

4.6 Fonctions et paramètres

L'élément de base de la programmation modulaire en C est la notion de fonction. Une fonction va réaliser des traitements (instructions) sur des données qui pourront être *locales globales* ou passées par l'intermédiaire de *paramètres* formels. Elle pourra par ailleurs renvoyer un résultat.

Une fonction est constituée d'un entête et d'un corps. On distingue la *déclaration* d'une fonction de sa *définition* bien que les deux puissent effectuées en même temps.

La *déclaration* d'une fonction consiste a indiquer:

- le type de la valeur retournée (void si la fonction ne retourne pas de valeur)
- son identificateur
- une liste de paramètres entre parenthèses séparés par des virgules avec pour chacun d'entre eux son type (cette liste peut être vide)

Comme une déclaration de variables, la déclaration d'une fonction est suivie d'un point-virgule (;).

Exemple:

```
long int puissance(long int, int);
```

ou encore:

```
long int puissance(long int x, int y);
```

La *définition* d'une fonction est composée d'une entête (semblable et compatible avec sa déclaration) et d'un bloc délimité par les caractères { et } contenant des déclarations de variables et des instructions.

Attention, une fonction ne peut pas contenir la déclaration d'une autre fonction.

L'instruction `return expression` définit la valeur qui est renvoyée par la fonction.

L'appel d'une fonction se fait simplement par son identificateur suivi de la liste des paramètres requis (entre parenthèses).

Exemples:

```
x=puissance(45,3);
y=puissance(x,2);
```

Si une fonction est *définie* avant son utilisation, il n'est plus nécessaire de la déclarer.

5. La lecture de la chaîne s'effectuera jusqu'au premier caractère “espace” rencontré.

4.7 Tableaux

Un tableau est une suite d'éléments de même type rangés consécutivement en mémoire.

On déclare un tableau en indiquant:

– le type des éléments qu'il contient.

– son identificateur

– le nombre d'éléments qu'il contient entre crochets.

Exemples:

```
int tableau[10]; /* tableau de 10 entiers */
float x[15];    /* tableau de 15 réels */
```

A chaque élément d'un tableau correspond un indice (valeur entière) qui permet d'accéder facilement à cet élément. L'indice d'un tableau en C commence à 0. Un tableau de N éléments contient donc des éléments de l'indice 0 à l'indice N-1

On accède à une valeur particulière du tableau en indiquant entre crochet l'indice de l'élément correspondant. Par exemple `x[3]` est la valeur du quatriéme élément du tableau `x`.

On peut initialiser un tableau au moment de sa déclaration en indiquant entre accolades la liste des valeurs de ses éléments.

Exemple:

```
int tableau[4]={10,5,67,34};
```

qui affecte à `tableau[0]` la valeur 10, à `tableau[1]` la valeur 5, etc...

4.7.1 Tableaux de caractères

Les chaînes de caractères sont stockées sous forme de tableaux de caractères le dernier élément de la chaîne devant être le caractère NULL. Une chaîne de caractères peut donc être déclaré de la manière suivante:

```
char chaine[10]; /* chaine de 9 caractères */
```

Attention, les tableaux ne sont pas des types de base du langage. On ne peut pas, par conséquent, effectuer des opérations globales dessus. Les opérateurs ne peuvent s'appliquer qu'élément par élément. En particulier, on ne peut pas comparer des chaînes de caractères entre elles à l'aide des opérateurs relationnels (==, !=,...). Il existe par contre des fonctions de bibliothèques standards qui permettent celà.

4.7.2 Tableaux multi-dimensionnels

Il existe aussi en C la possibilité de déclarer et d'utiliser des tableaux à plusieurs dimensions. Il s'agit en fait de tableaux de tableaux de tableaux... On accède à un élément du tableau en donnant un indice par dimension (entre crochets). On peut aussi initialiser ces tableaux au moment de la déclaration. Par exemple en dimension 2 la déclaration suivante crée et initialise un tableau de 2 lignes et 3 colonnes:

```
int matrice[2][3]={{1,2,3},{4,5,6}};
```

L'élément matrice[0][0] prend la valeur 1, L'élément matrice[0][1] prend la valeur 2, etc...

4.8 Pointeurs

4.8.1 Présentation

A une variable correspond un emplacement mémoire caractérisé par une adresse et une longueur (par exemple 4 octets consécutifs pour un `int`). C'est, bien sur, le compilateur qui assure la gestion

de la mémoire et affecte à chaque variable un emplacement déterminé. On peut accéder a la valeur de cette adresse grace à l'opérateur unaire & dit opérateur d'adressage.

Un pointeur est une variable d'un type spécial qui pourra contenir l'adresse d'une autre variable. On dit qu'il *pointe* vers cette variable. Celui-ci devra aussi connaître le type de la variable vers laquelle il pointe puisque la taille d'une variable (en octets) dépend de son type. La déclaration d'un pointeur devra donc indiquer le type d'objet vers lequel il pointe (on dit d'un pointeur qu'il est typé). La syntaxe est la suivante:

```
type *identificateur;
```

Par exemple, pour déclarer un pointeur vers un entier on écrira:

```
int* p_entier;
```

ou encore:

```
int *p_entier;
```

On peut réaliser des opérations sur des pointeurs de même type. On peut en particulier affecter à un pointeur un autre pointeur du même type ou l'adresse d'une variable de même type que celle du pointeur.

On accède a la valeur stockée à l'adresse contenue dans un pointeur grace à l'opérateur unaire dit de référencement ou d'indirection: *

Dans l'exemple suivant:

```
int a;
int* p_a;

p_a=&a;
```

`*p_a` et `a` font référence au même emplacement mémoire (et ont donc la même valeur).

Un pointeur peut par ailleurs pointer vers un autre pointeur.

On peut aussi incrémenter un pointeur. Celà revient a augmenter sa valeur de la taille de l'objet pointé et donc à pointer sur l'objet suivant du même type (s'il en existe un!).

La déclaration d'un pointeur alloue un espace mémoire pour le pointeur mais pas pour l'objet pointé. Le pointeur pointe sur n'importe quoi au moment de sa déclaration. Il est conseillé d'initialiser tout pointeur avant son utilisation effective avec la valeur NULL (constante prédéfinie qui vaut 0) ce qui par convention indique que le pointeur ne pointe sur rien.

4.8.2 Pointeurs et tableaux

La déclaration d'un tableau réserve de la place en mémoire pour les éléments du tableau et fournit une constante de type pointeur qui contient l'adresse du premier élément du tableau. Cette constante est identifiée par l'identificateur donné au tableau (sans crochets). C'est cette constante de type pointeur qui va permettre de manipuler facilement un tableau en particulier pour le passer en paramètre d'une fonction puisqu'on ne passera à la fonction que l'adresse du tableau et non tous ses éléments.

L'exemple suivant:

```
int tab[10];
```

déclare un tableau de 10 éléments. `tab` contient l'adresse du premier élément et donc l'expression:

```
tab == &tab[0]
```

est VRAIE. On a donc de même:

```
*tab == tab[0]
tab[1] == *(tab+1)
tab[k] == *(tab+k)
```

Les deux écritures sont autorisées.

La différence entre un tableau et un pointeur est qu'un tableau est une *constante* non modifiable alors qu'un pointeur est une *variable*.

4.8.3 Pointeurs et fonctions

On a vu que les paramètres passés à une fonction le sont par *valeur* et ne peuvent pas être modifiés par l'exécution de la fonction. Ceci est très contraignant si l'on souhaite qu'une fonction renvoie plusieurs résultats. Par exemple, si l'on souhaite écrire une fonction permuttant deux entiers, le code suivant qui parait correct ne fonctionnera pas:

```
void permutation(int a, int b)
{
   int c;

   c=a;
   a=b;
   b=c;
}
```

L'appel de la fonction:

```
permutation(x,y);
```

n'aura ainsi aucun effet sur x et sur y.

La solution consiste à passer, pour les paramètres que l'on souhaite voir modifier par la fonction, non plus les valeurs de ces paramètres mais les valeurs des adresses de ces paramètres.

Les paramètres de la fonction devront donc être des pointeurs. On accèdera aux valeurs propement dites à l'intérieur de la fonction gràce à l'opérateur d'indirection `*`.

Si l'on reprend l'exemple précédent, cela donne:

```
void permutation(int* p_a, int* p_b)
{
   int c;

   c=*p_a;
   *p_a=*p_b;
   *p_b=c;
}
```

Remarque: on aurait pu garder les identificateurs initiaux a et b!

Et l'appel devra se faire en passant en paramètres les adresses des variables à modifier gràce à l'opérateur d'adressage &:

```
permutation(&x,&y);
```

4.8.4 Portée des variables

En général, les déclarations de variables sont faites dans le corps des fonctions (y compris la fonction main). On parle de variables locales ou automatiques. Elles sont crées au début de l'exécution de la fonction et détruite quand celle-ci se termine.

Une fonction ne peut donc avoir accès à des varibles déclarées dans d'autres fonctions que par passage de paramètres. Ces paramètres contiennent une copie des valeurs passées en paramètre au moment de l'appel de la fonction.

Il est possible de déclarer des variables ayant une portée plus importante (variables globales). Ces variables doivent être déclarées en début de fichier et sont visibles par toutes les fonctions définies dans le fichier.

On peut aussi accéder à des variables globales définies dans d'autres fichiers à condition de les redéclarer avec le mot clé `extern`.

4.9 Structures

Une structure rassemble des variables, qui peuvent être de types différents, sous un seul nom ce qui permet de les manipuler facilement. Elle permet de simplifier l'écriture d'un programme en regroupant des données liées entre elles.

Un exemple type d'utilisation d'une structure est la gestion d'un répertoire. Chaque fiche d'un répertoire contient (par exemple) le nom d'une personne, son prénom, son adresse, ses numéros de téléphone, etc... Le regroupement de toutes ces informations dans une structure permet de manipuler facilement ces fiches.

Autre exemple: On peut représenter un nombre complexe à l'aide d'une structure.

On définit une structure à l'aide du mot-clé: `struct` suivi d'un identificateur (nom de la structure) et de la liste des variables qu'elle doit contenir (type et identificateur de chaque variable). Cette liste est délimitée par des accolades. Chaque variable contenue dans la structure est appelée un *champ* ou un *membre*.

Définir une structure consiste en fait à définir un nouveau type. On ne manipulera pas directement la structure de la même manière que l'on ne manipule pas un *type*. On pourra par contre définir des variables ayant pour type cette structure.

Exemple:

```
struct complexe
  {
    double reel;
    double imag;
  };

struct complexe x,y;
```

Dans cet exemple, on définit une structure contenant deux réels puis on déclare deux variables ayant pour type `struct complexe`.

Les opérations permises sur une structure sont l'affectation (en considérant la structure comme un tout), la récupération de son adresse (opérateur &) et l'accès à ses membres.

On accède a la valeur d'un membre d'une structure en faisant suivre l'identificateur de la variable de type structure par un point suivi du nom du membre auquel on souhaite accéder. Par exemple `x.reel` permet d'accéder au membre 'reel' de la variable x.

On peut initialiser une structure au moment de sa déclaration, par exemple:

```
struct complexe x={10,5};
```

On peut définir des fonctions qui renvoie un objet de type structure. Par exemple, la fonction suivante renvoie le complexe conjugué de x:

```
struct complexe conjugue(struct complexe x);
{
  struct complexe y;

  y.reel=x.reel;
  y.imag=-x.imag;
```

```
  return y;
}
```

L'utilisation de cette fonction pourra ressembler à:

```
struct complexe x,z;

z=conjugue(x);
```

L'affectation revient à affecter chaque membre de la structure, par exemple:

```
struct complexe x,z;

x=z;
```

est équivalent à:

```
struct complexe x,z;

x.reel=z.reel;
x.imag=z.imag;
```

4.9.1 Pointeurs de structure

Une structure peut être un objet complexe possédant de nombreux membres. Le passage d'un objet à une fonction revient à faire une copie en mémoire de cet objet. Il peut être plus avantageux de passer simplement l'adresse de cet objet auquel cas c'est seulement une copie de l'adresse qui est effectuée. Le passage par adresse est aussi nécessaire si une modification de la valeur d'un ou plusieurs membres de la structure doit être modifiée par la fonction.

L'accès au contenu d'une variable de type pointeur se fait grace a l'opérateur de référencement *. C'est la même chose pour un pointeur de structure mais avec un raccourci de notation pour accéder a un membre de la structure: Exemple:

```
struct complexe * x; /* pointeur de structure */
struct complexe y;

y.reel=(*x).reel; /* accès au champ 'reel' */
y.reel=x->reel;   /* même chose, seule la notation change */
```

4.9.2 Définition de types

Le C fournit un fonctionnalité qui permet de créer de nouveaux noms de type.

Par exemple:

```
typedef  int Entier;

typedef struct complexe Complexe;
```

4.10 Fichiers (Entrées/Sorties)

Nous avons vu précédemment les entrées/sorties clavier/écran. Nous allons voir que ce n'était qu'un cas particulier des mécanismes de manipulation de fichiers.

4.10.1 Le type `FILE*`

Les fonctions de manipulation de fichiers font parties de la bibliothèque d'entrées/sorties standards: *stdio*. C'est dans cette bibliothèque qu'est défini le type `FILE*`.

Une variable de type `FILE*` est un descripteur de fichier ou encore un pointeur de fichier. On parle aussi de flux (stream en anglais).

Trois descripteurs de fichiers sont prédéfinis et disponibles dès qu'un programme fait appel à la, bibliothèque *stdio* grace à la directive de compilation

```
#include <sdtio.h>
```

Ce sont, respectivement:

- `stdin` entrée standard (le clavier)
- `stdout` sortie standard (l'écran)
- `stderr` sortie pour les messages d'erreur (l'écran)

L'accès à un fichier sur disque se fait en:

- déclarant une variable de type `FILE*`
- en initialisant cette variable à l'aide de la fonction `fopen`

4.10.2 Ouverture d'un fichier

Tout fichier doit être ouvert avant d'être utilisé (excepté stdin, stdout et stderr qui le sont implicitement). C'est la fonction `fopen` qui permet de réaliser cette opération. La syntaxe est la suivante:

```
FILE* fopen(char* nom, char* mode)
```

Le premier argument est une chaîne de caractères contenant le nom du fichier. Le second argument est aussi une chaîne de caractères et contient le mode d'ouverture. Celui-ci peut être une ouverture en lecture (r) en écriture (w) ou en ajout (a).

Si l'on essaie d'ouvrir en lecture un fichier qui n'existe pas la fonction fopen() renvoie la valeur NULL.

Si on ouvre en écriture un fichier qui n'existe pas, celui-ci est créé.

Si on ouvre en écriture un fichier qui existe, le contenu de celui-ci est écrasé.

Il est conseillé de vérifier que l'opération d'ouverture s'est bien passé avant d'essayer d'accéder au fichier. Par exemple, on écrira:

```
FILE* fichier;

fichier=fopen("donnees.dat","r");
if (fichier==NULL)
  {
    printf("erreur dans l'ouverture du fichier\n");
    exit(1);
  }
```

La fonction exit() permet de sortir du programme en renvoyant un code. Par convention, on renvoie une valeur différente de 0 en cas d'erreur.

4.10.3 Fermeture d'un fichier

Un fichier doit être fermé quand on n'a plus besoin d'y accéder. C'est la fonction `int fclose(FILE* f)` qui permet d'effectuer l'opération. Elle prend en argument le nom d'un descripteur de fichier.

4.10.4 Entrées-sorties de caractères

Lecture d'un octet (le caractère EOF est renvoyé en fin de fichier):
`int getc(FILE* fichier);`

`c=getc(sdtin);` est équivalent à `c=getchar();`

Ecriture d'un octet:
`int putc(char c, FILE* fichier);`

`put(c,stdout);` est équivalent à `putchar(c);`

4.10.5 Entrées-sorties formatées

Lecture:
`int fscanf(FILE* fichier, char* format, &arg1, &arg2,...)`

La signification de "format" est la même que pour une lecture au clavier (fonction `scanf`).

La fonction *fscanf* renvoie la valeur EOF (int) à la fin du fichier. Par exemple pour lire des nombres réels jusqu'a la fin d'un fichier et les ranger dans un tableau on écrira:

```
float nombre[50];
FILE* f;
int fin;
int n=0;

f=fopen("donnees.dat","r"); /* ouverture en lecture */

fin=!EOF;
while(fin!=EOF)
 {
  fin=fscanf(f,"%f ",&nombre[n]);
  n++;
 }
fclose(f);
```

Ou de manière plus concise:

```
while(fscanf(f,"%f ",&nombre[n++])!=EOF);
```

Ecriture:
`int fprintf(FILE* fichier, char* format, arg1, arg2,...)`

La signification de "format" est la même que pour une écriture à l'écran (fonction `printf`).

5 Compléments

5.1 Directives de compilation

Les directives de compilation sont des lignes traduites par le pré-processeur. Elles commencent par le symbole #.

- `#include <nom_fichier.h>`
  ou `#include "nom_fichier.h"` permet d'inclure un fichier d'entêtes (i.e. contenant des déclarations de fonctions, de constantes, de types, et/ou de variables globales). Le fichier est recherché dans le répertoire courant ou dans un répertoire spécifique.

– `#define NOM texte` Les occurences de “NOM” sont remplacées par le texte qui suit (jusqu'à la fin de ligne)..
– `#if, #else, #elif, #endif` Compilation conditionnelle. Si l'expression logique qui suit le if est VRAI (différente de 0), les lignes qui suivent (jusqu'au `#endif`) sont compilées, sinon elles sont ignorées. L'expression `define(NOM)` est vrai si NOM est déja défini, fausse sinon. On peut aussi utiliser les formes spécialisées qui testent si NOM est défini ou pas: `#ifdef NOM` et `#ifndef NOM`.

Exemple d'utilisation de `#define`:

```
#define N 10

int tableau[N];
...
for (i=0; i<N; i++) tableau[i]=...
...
```

5.2 Paramètres de la ligne de commande

Il est possible de passer des arguments à un programme au début de son exécution. En effet la syntaxe complète de la fonction `main` est:

```
int main(int argc, char* argv[])
```

Celle-ci possède donc deux paramètres:
– Un entier: `argc` qui contient le nombre d'arguments de la ligne de commande qui a permis de lancer le programme.
– Un tableau de chaînes de caractères `argv[]`, chaque chaîne du tableau contenant un argument passé au programme. Par défaut `argv[0]` contient le nom du programme (exécutable) lui-même.

Par exemple, la ligne de commande suivante:

```
monprog toto tata 45
```

Met 4 dans `argc`, `"monprog"` dans `argv[0]`, `"toto"` dans `argv[1]`, `"tata"` dans `argv[2]`, et `"45"` dans `argv[3]`.

Ces variables peuvent être ensuite utilisées dans le programme comme les autres variables déclarées dans la fonction `main`.

Pour convertir les chaînes de caractères en entier ou en réel, il existe des fonctions définies dans la librairie `stdlib.h` décrite plus loin.

5.3 Allocation dynamique

La déclaration de variables dans la fonction `main` ou globalement réserve de l'espace en mémoire pour ces variables pour toute la durée de vie du programme. Elle impose par ailleurs de connaître avant le début de l'exécution l'espace nécessaire aux stockage de ces variables et en particuliers la dimension des tableaux. Or dans de nombreuses applications le nombre d'éléments d'un tableau peut varier d'une exécution du programme à l'autre.

La bibliothèque `stdlib` fournit des fonctions qui permettent de réserver et de libérer de manière dynamique (à l'exécution) la mémoire.

La fonction qui permet de réserver de la mémoire est `malloc` [6] Sa syntaxe est:

```
void* malloc(unsigned int taille)
```

6. il existe aussi la fonction `calloc` assez similaire et donc superflu...

La fonction malloc réserve une zone de *taille* octets en mémoire et renvoie l'adresse du début de la zone sous forme d'un pointeur non-typé (ou NULL si l'opération n'est pas possible).

En pratique, on a besoin du type d'un pointeur pour pouvoir l'utiliser. On souhaite d'autre part ne pas avoir à préciser la taille de la mémoire en octets surtout s'il s'agit de structures. L'usage consiste donc pour réserver de la mémoire pour une variable d'un type donné à:

- Déclarer un pointeur du type voulu.
- Utiliser la fonction `sizeof(type)` qui renvoie la taille en octets du type passé en paramètres.
- forcer `malloc` à renvoyer un pointeur du type désiré.

Par exemple, pour réserver de la mémoire pour un entier, on écrira:

```
int* entier;

entier=(int*) malloc(sizeof(int));
```

Ceci est surtout utilisé pour des tableaux, par exemple pour un tableau de N "complexe" (cf. le paragraphe sur les structures) on écrirai:

```
struct complexe * tabcomp;

tabcomp=(struct complexe*) malloc(N*sizeof(struct complexe));
```

La fonction `free()` permet de libérer la mémoire précédemment réservée. Sa syntaxe est:

```
void free(void* p)
```

Ainsi, dans le second exemple on écrirai:

```
free(tabcomp);
```

5.4 Quelques bibliothèques standards

Nous décrivons ci-après quelques fonctions (les plus utilisées) de quelques bibliothèques standards. Il ne s'agit donc pas d'une présentation exhaustive (loin de là...).

5.4.1 math.h

L'inclusion du fichier `math.h` permet d'accéder à un certain nombre de fonctions mathématiques.

- `double fabs(double x)` Valeur abolue de x
- `double exp(double x)` Exponentielle de x
- `double log(double x)` Logarithme népérien de x
- `double log10(double x)` Logarithme décimal de x
- `double pow(double x, double y)` x à la puissance y
- `double sqrt(double x)` Racine Carrée de x
- `double sin(double x), double cos(double x), double asin(double x), double acos(double x), double atan(double x)` Fonctions trigonométriques (x en radians) [7].
- `double sinh(double x), double cosh(double x), double tanh(double x)` Fonctions hyperboliques.

Il est nécessaire d'indiquer que l'on utilise la librairie mathématique lors de la compilation du programme. Ceci est fait en rajoutant l'option -lm, par exemple:

```
gcc -Wall -o prog prog.c -lm
```

7. la valeur de π peut être obtenue grace à `2*asin(1)`

5.4.2 stdlib.h

Cette bibliothèque fournit un certain nombre de fonctions utilitaires:

- `double atof(char *)` Conversion d'une chaîne en réel.
- `int atoi(char *)` Conversion d'une chaîne en entier.
- `void* malloc(unsigned int taille)` réserve de l'espace en mémoire.
- `void free(void* p)` libére de l'espace mémoire.
- `void srand(unsigned int seed)` fixe le début d'une séquence pseudo aléatoire.
- `int rand()` retourne un entier pseudo aléatoire compris entre 0 et RAND_MAX (constante prédéfinie) suivant une loi uniforme.

5.4.3 string.h

L'inclusion du fichier `string.h` permet d'accéder a des fonctions de manipulation de chaînes de caractères. Les principales sont les suivantes:

- `char* strcpy(char* s, const char* ct)` copie la chaîne ct dans s et renvoie s.
- `char* strcat(char* s, const char* ct)` concatène la chaîne ct à la suite de la chaîne s et renvoie s.
- `int strcmp(const char* cs, const char* ct)` compare la chaîne ct et la chaîne cs. Renvoie une valeur nulle si $cs == ct$, négative si $cs < ct$ et positive si $cs > ct$.
- `char* strchr(const char* cs, char c)` renvoie un pointeur sur la première occurence de c dans cs (renvoie NULL si pas d'occurence).
- `char* strstr(const char* cs, const char* ct)` renvoie un pointeur sur la première occurence de t dans cs (renvoie NULL si pas d'occurence).